AN ANALYSIS OF RULE INDEXING IMPLEMENTATIONS IN DATA BASE SYSTEMS

M. Stonebraker, T. Sellis and E. Hanson

University of California
Berkeley, Ca.

Abstract

In this paper we discuss several alternate implementation schemes for rule indexing in a data base system. Two of the proposals have much in common with predicate locking, while four others resemble versions of physical locking. A performance analysis is conducted based on an abstract model of the rule indexing problem.

1. INTRODUCTION

There has been considerable discussion of the relative merits of predicate locking [ESWA76] and physical locking [GRAY78] to support concurrency control in relational data base systems. Physical locking has been implemented in most commercial relational systems with which we are familiar, and appears to be the tool of choice.

In this paper we show that similar considerations arise when a relational DBMS is extended to support rule processing for expert system applications. In this new context, tactics similar to both predicate locking and physical locking can be applied, and it is necessary to re-examine the best choice in light of the changed circumstances.

In Section 2 we indicate three different environments in which rule processing must be accomplished. Each will be seen to require very similar functionality. Then we turn in Section 3 to a discussion of support for these environments which use tactics similar to physical locking and predicate locking. Section 4 indicates an abstract model with which we can compare the performance of the two approaches. Section 5 then discusses predicted performance of the various implementations in a collection of different situations. Lastly, we conclude in Section 6 by discussing a collection of more sophisticated algorithms which may offer superior performance compared to the ones analyzed.

2. RULE PROCESSING ENVIRONMENTS

2.1. Triggers

The first environment of interest is support for active data bases which include triggers. One possible syntax was presented in a previous paper [STON85a], and others have been designed (e.g. [ESWA75]). We will use the folowing schema for the standard EMP relation for examples in this paper:

relation: EMP(name, age, salary, status)
storage structure: B^+-tree indexes on name, salary

This research was sponsored by the U.S. Air Force Office of Scientific Research Grant 83-0254 and by the National Science Foundation under Grant xxxxxx.

A trigger to set Mike's salary equal to Sam's salary whenever Sam is given a raise would be expressed as follows:

range of E is EMP
replace ALWAYS EMP (salary = E.salary)
where EMP.name = "Mike"
and E.name = "Sam"

The required semantics are that this command should logically appear to run indefinitely. More realistically, whenever a command such as

replace EMP (salary = 1000) **where** EMP.name = "Sam"

is processed the trigger should be awakened to update Mike's salary.

More precisely, the system must store a (perhaps very large) collection of triggers:

T_1: **replace** ALWAYS relname-1 (Target-list-1) **where** PREDICATE-1
.
.
.
T_n: **replace** ALWAYS relname-n (Target-list-n) **where** PREDICATE-n

When a user update (i.e. a replace or an append) is processed, e.g.

update-command relname (Target-list) **where** QUAL

the system must find all triggers T_i, for which there exists a tuple *t* modified or inserted by the update such that:

t satisfies PREDICATE-i
and
t satisfies QUAL
and
Target-list contains an attribute which appears in Target-list-i or PREDICATE-i

Notice, that the only penalty for finding "false drops" (i.e. T_i for which the above condition is not true) is that time is wasted processing a trigger which does not actually do anything.

2.2. Inference

One possible approach to inferring data which is not present in the data base was also presented in [STON85a]. In that proposal columns of a relation are either real or virtual. Real fields are filled with normal data while virtual fields are inferred from rules associated with the relation. One possible rule would be:

replace DEMAND EMP (salary = E.salary)
where EMP.name = "Mike"
and E.name = "Sam"

This rule states that the salary of Mike should be the same as the salary of Sam. A second rule might require Sam's salary to be 1000 dollars as follows:

replace DEMAND EMP (salary = 1000)
where EMP.name = "Sam"

One wishes an environment whereby a user can ask queries of the form:

retrieve (EMP.salary) **where** EMP.name = "Mike"

and have the system infer the correct answer of 1000 from the above rules.

The virtues of this approach relative to utilizing the view mechanism to perform inference (e.g. [ULLM85, IOAN84]) have been presented in [STON85a]. In brief, the view mechanism is appropriate when a small collection of rules is present and most of them are applicable. The canonical example is the

construction of the ANCESTOR view from a base PARENT relation. On the other hand, if there are many rules which define a particular view and most are not relevant to a given query, then our proposal excells. Moreover, our proposal supports conflicting rules. For example, a third rule might state that all managers must be paid 2000 dollars, e.g:

replace DEMAND EMP (salary = 2000) **where** EMP.status = "mgr"

If Mike is promoted from a worker to a manager, then his salary should be changed from 1000 to 2000. This requires a mechanism whereby multiple rules can apply in a given situation and one is designated at higher priority to be used in preference to the others. Such priority situations are common in AI applications and are straightforwardly supported in the scheme of [STON85a] but are much more difficult in rule systems which utilize the view processing system.

This environment requires storing a (perhaps very large) collection of DEMAND commands:

D_1: **replace** DEMAND (Target-list-1) **where** PREDICATE-1
.
.
.
D_n: **replace** DEMAND (Target-list-n) **where** PREDICATE-n

Then, the system must process a user command:

retrieve (Target-list) **where** QUAL

by finding all D_i for which there exists a tuple *t* such that:

t satisfies PREDICATE-i
and
t satisfied QUAL
and
Target-list intersect Target-list-i NON-EMPTY

For qualifying D_i, one must run an algorithm similar to query modification [STON75] to convert the original user retrieval into a new one which is then processed. The details of this algorithm appear in [STON85a]. Again, there is no penalty for "false drops", as they simply generate excess overhead.

2.3. Precomputed Answers to Commands

A third environment with similar requirements is one which allows queries in the query language to be data base objects [STON84]. For example, one could declare the salary field of the EMP relation to be a query language command. The value for this field for the employee named Mike would be:

range of E is EMP
retrieve (salary = E.salary) **where** E.name = "Sam"

Of course, Sam's salary would be the query returning a constant:

retrieve (salary = 1000)

The difference between this model and the previous one is that queries are bound to specific data tuples, and the value of the field is found by executing a query. For example, if all the employees in the shoe department share the same salary, then the query specification must be repeated for each one. On the other hand, the previous environment did not bind a query to a specific tuple, and a single DEMAND command could apply to all employees in the shoe department.

In this situation a desirable optimization strategy is to optionally cache the answer to queries which are executed [STON85b]. The next time one accesses the object, the data is already precomputed and need not be rematerialized. However, the system must invalidate the precomputed object (e.g the salary of Mike) if a subobject from which it is composed (e.g. the salary of Sam) is updated.

More precisely, this environment contains a collection of queries which have been precomputed, e.g:

Q_1: **retrieve** (Target-list-1) **where** PREDICATE-1

.
.
.

Q_n: **retrieve** (Target-list-n) **where** PREDICATE-n

If an update command (a replace or append) is processed, e.g:

update-command (Target-list) **where** QUAL

the system must ascertain which retrieve commands must be invalidated. The required test is to find all Q_i for which there exists a tuple, *t* such that:

t satisfies PREDICATE-i
and
t satisfies QUAL
and
Target-list contains an attribute which appears in Target-list-i or PREDICATE-i

Notice that this test is identical to the trigger test.

3. SUPPORT FOR RULE PROCESSING

The first option for supporting rule processing would be to construct a theorem prover which would find all PREDICATE-*i* for which:

QUAL AND PREDICATE-*i* NOT EMPTY

Analysis of such a theorem prover is beyond the scope of this paper, and we restrict our attention to more conventional tactics. Also inference requires tuple-by-tuple processing on retrieves and bulk processing is not applicable. Hence, in the remainder of this paper we assume that tuples satisfying QUAL are determined in the course of normal query processing. For each such tuple, *t* the task is then the following:

Find all rules, D_i, T_i or Q_i depending on the environment, such that *t* satisfies PREDICATE-*i*.

Also, depending on the type of rules being used, there may be additional checking concerning presence of attributes.

The first approach which we consider is to view each PREDICATE-*i* as setting a predicate lock in the data base. When the tuple *t* is accessed, one must ascertain which predicate locks cover it. When the number of predicates is large, it is best to construct a predicate index to avoid a sequential search of all predicates. Given such an index, to find the predicates covering a tuple, one can perform an index lookup. In Section 3.2 we present a predicate indexing scheme for rule processing that has points in common with predicate locking proposals.

Alternatively, one can view each predicate as setting physical locks. Since each PREDICATE-*i* is a valid query qualification, a database command corresponding to it can be executed by the query processing engine. When this command executes, a special marker can be set on each accessed tuple instead of a conventional read or write lock. Such a marker, called a ‘‘trigger-me lock’’ or ‘‘t-lock,’’ indicates that PREDICATE-*i* *might* cover the tuple. When the collection of rules that cover a tuple, *t* is required, these locks can be consulted. We present an algorithm based on t-locks in Section 3.3.

Then, in Sections 3.4 - 3.6 we discuss four other variants of the basic predicate locking and physical locking techniques. These alternatives may prove attractive in certain situations.

3.1. Predicate Indexing

The goal of this scheme is to build a data structure that will allow the rule base to be efficiently searched to determine the PREDICATE-*i* that cover a specific tuple. In this section we assume that all PREDICATE-*i* have the following characteristics:

1) each PREDICATE-*i* restricts a single relation and contains a single tuple variable

2) each PREDICATE-*i* is a conjunction of terms of the form:

constant-1 ≤ attribute-i ≤ constant-2

for 1 ≤i ≤F.
Constant-1 and constant-2 may be scalars or positive
or negative infinity.

Notice, that this formulation does not allow the presence of join clauses in predicates. Moreover, the indexing proposal in this section cannot easily be extended to capture join predicates.

For each relation that has rules defined, we propose to build a special kind of R-tree [GUTT84a] on the total of F attributes that may have a restrictive term in a predicate. We select R-trees instead of some other multidimensional structure like k-D-B-trees [ROBI81] because rules do not represent point data in a multidimensional space, but rather rectangular regions. For example,

20 < EMP.age < 30 and 10K < EMP.salary < 30K

defines a rectangle in the two dimensional space with dimensions EMP.age and EMP.salary.

An R-tree is a tree structure used to index rectangles in a geometric environment [GUTT84b]. It is clear that the predicates indicated above can be considered rectangles in the multidimensional space formed by considering each of the F attributes as a dimension. An intermediate node in an R-tree is a sequence of pairs (*RECT*,*PTR*), where *RECT* is the description of a rectangle and *PTR* a pointer. The pointer *PTR* is used to point to another node all of whose rectangles are completely contained in *RECT*. The leaf nodes have the same format, except that there are no pointers.

One major difference between R-trees and B-trees is that the rectangles, *RECT* found in intermediate nodes of an R-tree are not disjoint. Therefore, when descending the tree searching for the predicates which cover a specific tuple, t, one may investigate more than one path in the R-tree. When the indexed rectangles are large, as may be the case in this application, considerable overlap of bounding rectangles may be observed, and many paths may require investigation.

In this application, an alternate variation of R-trees may prove attractive. Any rule predicate can be decomposed into a collection of non-overlapping sub-rectangles whose union is the original predicate. These sub-rectangles can be judiciously chosen so that no bounding rectangle in any index level of the R-tree need be enlarged. Moreover, when a leaf-level page overflows and the page must be split, the bounding rectangle can usually be partitioned into two non-overlapping rectangles and then the sub-rectangles on the page which intersect the new boundary can be split. However, in the worst case where there are n rules that all mutually intersect, then a sub-rectangle of each rule will occur on some leaf page. If the capacity of this page is exceeded, a split will not lower its occupancy and the leaf page must be extended with overflow pages.

In this case, only a single path in the tree must be investigated to find the predicates covering t. The cost is a perhaps much larger number of rules to index. We will call the new tree type, R^+-trees, and a detailed investigation of its characteristics is presented in [SELL86]. In the performance comparison that follows in Section 4, we analyze only R^+-trees because we can only find closed form expressions for expected costs in this case. Which variation actually performs better depends on the composition of the predicates.

It should be clear that one must descend an R^+-tree from the root to one leaf node every time one wishes to find the predicates that cover a specific tuple. However, one need do no special maintenance of the tree when a tuple is inserted into the database or modified.

3.2. Basic Locking

In this scheme, a relation

RULES(id, rule-def)

will contain the rule base. The *id* field contains a unique identifier for the rule, and *rule-def* contains the definition of the rule, including its predicate.

For each rule which is defined, an access plan is constructed by the query optimizer. This plan is executed and each tuple it reads is marked with a t-lock containing an identifier for the current predicate. If a sequential scan of the relation is used, then all tuples in the relation will be marked. In this case we assume that conventional lock escalation will convert record locks to a relation lock. Otherwise, an index will be used for access and t-locks will be set on data records *and* on the key interval inspected in the index. Such index interval locks are required to correctly deal with insertion of new records, as explained momentarily. The exact form of the index locks may be specific to the type of index, and our analysis in Section 4 assumes that t-locks can be set between keys on the leaf level of the index. To simplify the analysis we also require that all predicates contain at least one indexed attribute, thus avoiding the use of any relation-level locks.

If a tuple *t* is inserted, then the collection of markers must be found for the new tuple. As a side effect of the insertion, values will be inserted into various indexes. If such a value is covered by a key-range lock, then a corresponding t-lock will be added to the data tuple containing the value. If a tuple is modified, it should be viewed as a deletion followed by an insertion.

To ascertain what collection of PREDICATE-*i* cover a tuple *t*, one first collects all the t-locks on *t*. Since these t-locks represent a superset of the predicates that actually match the tuple, relevant tuples in the RULES relation must be checked to determine whether *t* actually satisfies each one.

For example, the qualification:

EMP.salary = 1000 and EMP.age > 30

will set t-locks in the salary index and on all data records that it reads (i.e. those with salary = 1000). Not all of these will have qualifying ages. The reason such a cautious strategy must be adopted is that a non-indexed attribute may be modified so that a record matches a predicate it did not match before the change. For example, an employee may be aged from 30 to 31. Since there is no secondary index on age, the basic algorithm would have no way of discovering that it should now be marked, barring searching the salary index, a cost we wish to avoid if only age is updated. Because of this problem, t-locks must be set on all tuples that *potentially* satisfy a predicate based on the interval locks the predicate has set in the indexes.

This strategy is called basic locking because it sets t-locks on all objects for which a normal query would set read or write locks. Moreover, it requires no changes to conventional execution of access plans, so it can be properly called a locking mechanism. The advantage of this scheme is that it is closely coupled to normal query processing. New qualifications can be added using normal facilities, and locks for new tuples are found as byproducts of normal update processing.

In the next three sections we indicate variations on basic predicate locking and basic physical locking.

3.3. Early Basic Locking

Notice that the above algorithm defers checking whether a given tuple actually satisfied a predicate until the tuple is accessed. The algorithm does only a modest amount of work at the time a tuple is inserted or modified, leaving the bulk of the overhead to the time the tuple is accessed. Hence, the algorithm can be classified as a "late" algorithm, in that it defers overhead whenever possible. If tuples are updated frequently, late algorithms should perform well. On the other hand, if a tuple is accessed often, the following "early" locking algorithm should prove advantageous.

In early basic locking, a collection of t-locks is constructed at the time a tuple is inserted or modified just as in basic locking. However, an extra step is also performed, namely each corresponding predicate from the RULES relation is accessed and checked against the tuple. T-locks are only stored on the tuple for predicates which the tuple satisfies and the "false drops" are eliminated. In exchange for extra overhead at update time, this algorithm can deliver a list of qualifying rule-ids with no extra overhead at the time of access.

3.4. Early Predicate Indexing

An analogous "early" version of predicate indexing can also be constructed. In this situation, ascertaining which predicates cover a given tuple is not deferred to access time; rather it is done at insertion time. Hence, markers for the predicates which actually cover the tuple are stored on the tuple as in early basic locking. However, these markers are discovered by accessing the predicate index rather than by consulting index interval locks in appropriate indexes. Again, this scheme should be advantageous in "read mostly" environments.

3.5. Boolean Basic Locking

This scheme will be appropriate for query processing engines which use all applicable indexes when constructing a query plan. Hence, when the query plan for any predicate is executed *all* possible indexes are consulted and lists of record identifiers are intersected or merged before any data tuples are accessed. Then, only those tuples which satisfy all indexed predicates are actually read. In this scheme, index interval locks will be set in *all* possible indexes in contrast to basic locking where index locks will be set in only one index. In addition tuple locks will be set on only those tuples which satisfy *all* indexed predicates in contrast to basic locking where locks will be set on all tuples satisfying the predicate(s) for one indexed attribute.

This scheme will have a lesser number of false drops than basic locking, but will require setting a larger number of index interval locks. Also, correctness of this algorithm requires that the query planner be forced to consult all indexes. Notice lastly that an "early" version of boolean locking can be defined.

In order to analyze the performance of the six algorithms discussed so far:

(late) basic locking
early basic locking
(late) predicate indexing
early predicate indexing
(late) boolean basic locking
(early) boolean basic locking

we now define an abstract model for the operations performed on the database. Through this model we will be able to analyze the performance of the above implementations in various contexts.

4. THE ABSTRACT MODEL

The schemes discussed so far have dissimilar characteristics. Late predicate indexing requires all requests for the set of covering predicates to go through the index while early basic locking delivers the collection that qualify directly. When tuples are updated or inserted, there is no extra overhead paid by the "late" schemes while all "early" methods incur a substantial penalty. This section defines an abstract model of relevant operations so that performance of the various algorithms can be analyzed.

4.1. The Abstract Model

We assume that there are two relevant operations in the model. The first is to update a single field in a single tuple in a relation R which has F attributes. This single field is chosen at random from among the F candidates. The second is to find all the predicates which cover a given tuple which already exists in the data base. This tuple is specified by giving a key value as a qualification. The reason for this model is that most data base operations can be built up from these primitives. For example, an insert is approximately modeled by one replace operation for each attribute. In addition, a retrieve operation is a collection of basic tuple retrieves.

Moreover, the three applications presented in Section 2 are composites of these two operations. For example, the trigger environment requires tuples to be identified, all predicates which cover these tuples to be found and then the proposed update to be performed. This can be considered as a collection of pairs of basic operations in our model, each containing a retrieve followed by an update. The other two example applications are similarly modeled.

The mix between updates and retrieves is controlled by a parameter, P, which is the percentage of updates. The other parameters used in the analysis are presented in the following table.

Parameter	Description
C_1	The cost of evaluating a predicate for a given tuple
C_2	The cost of reading a page
B	The size of the page in bytes
W	The width of data records in the relation
N	The number of tuples in the relation
F	The number of fields in the relation
F_I	The number of fields in the relation with an index
S	The width of individual fields in the relation
4	The assumed width of pointers and t-locks
t	The number of rules
Q	The fraction of records matching a single term of a predicate (will vary with the model)
P	The fraction of update commands

The Model Parameters
Table 1

Moreover, we will analyze each of the following four models for the rule predicates:

Model 1: All predicates have a single clause restricting a single field z, i.e. they are of the form

relation.z = value

Model 2: All rule predicates are of the form

lvalue ≤ relation.z ≤ uvalue

Model 3: Each predicate has an equality restriction clause on all of the F attributes in the relation.

relation.z_1 = value$_1$ **and** ... **and** relation.z_F = value$_F$

Model 4: This model is the same as model 3, except that the individual clauses are range restrictions rather then exact-match terms.

lvalue$_1$ ≤ relation.z_1 ≤ uvalue$_1$ **and** ... lvalue$_F$ ≤ relation.z_F ≤ uvalue$_F$

We now turn to the expected cost per operation for late predicate indexing, late basic locking, and late boolean basic locking for each of the four predicate models. The expected costs of the early versions of these implementations are straight forward to derive, and we omit them in the interest of brevity. In all cases, we assume that the general algorithm is utilized, and that special case features appropriate to a particular predicate class cannot be exploited. This corresponds to a realistic implementation which does not know the composition of the rules in advance. Also, we only count processing costs in excess of those required by any system to perform the retrieval and update operations.

4.2. Late Predicate Indexing

The predicate index must be built on all F attributes of the relation. Assuming 4 bytes for each pointer and W to be the width of a tuple, then each node can hold

$$p = \lfloor \frac{B}{2W + 4} \rfloor$$

predicates. We will also assume for simplicity that all nodes are full. (Otherwise one can use some constant factor such as the one derived in [YAO78]). In R^+-trees the rules may broken into more than one entry in models 2 and 4, and a reasonable approximation of the expected number of smaller pieces O that a rule must be broken into is

$$O = \begin{cases} 1 & \text{for models 1 and 3} \\ 2 & \text{for model 2} \\ 2^{F/2} & \text{for model 4} \end{cases}$$

Given the above assumptions, the number of leaf pages, L in the index is:

$$L = \lceil \frac{t \cdot O}{p} \rceil$$

and the depth of the index is:

$$d = lceil \log_p Lrceil$$

In this case the cost to find all predicates that overlap a specific tuple is, for models 1 and 3

$$cost = (d + 1)(C_2 + C_1 \cdot p)$$

while for models 2 and 4

$$cost = (d + 1)(C_2 + C_1 \cdot p) + (t \cdot Q - p)C_1 + lceil \frac{t \cdot Q - p}{p} rceil C_2 \quad \text{if } t \cdot Q > p$$

$$cost = (d + 1)(C_2 + C_1 \cdot p) \quad \text{if } t \cdot Q \leq p$$

and the total cost per operation, TOTAL in the abstract model is:

$$\text{TOTAL} = (1 - P)cost$$

4.3. Late Basic Locking

It is clear that an insert incurs zero extra overhead for this algorithm. The only cost occurs in finding covering predicates. The predicates corresponding to all t-locks must be accessed (at cost C_1 each) and then checked (at cost C_2) to find the ones which actually cover the tuple. Therefore, the total cost per operation in the abstract model is:

$$\text{TOTAL} = (1 - P)t \cdot Q \cdot (C_1 + C_2)$$

4.4. Late Boolean Basic Locking

It is clear that there will be no false drops for any of the predicates, since all have an indexed attribute. Hence, the only extra work will occur during insert operations. The number of markings which will be collected and checked in model 1 and 2 is:

$$num = t \cdot Q$$

while in models 3 and 4 we have to check $t \cdot Q$ markings from each of the F_I indexes, that is

$$num = F_I \cdot t \cdot Q$$

markings. The total cost of this operation is:

$$cost_1 = num \cdot (C_1 + C_2)$$

Moreover, each secondary index which is defined over attributes used in predicates must be accessed even if it is not being updated. All indexes have N attribute values and pointers to N records. However, in models 1 and 2, only the index on attribute z has t-locks while the rest do not. In models 3 and 4 all indexes have t-locks.

For attribute z in models 1 and 2, there are $t \cdot Q$ t-locks in the index which consume $4 \cdot t \cdot Q$ bytes. Assuming that all index pages are full, the number of leaf pages, L in the index is:

$$L = \lceil \frac{4 \cdot t \cdot Q + 4 \cdot N + S \cdot N}{B} \rceil$$

and the depth of the index is

$$d = lceil \log_f Lrceil$$

where the fanout f is given by:

$$f = \lfloor \frac{B}{S+4} \rfloor$$

The cost of processing this index is:

$$cost_2 = (d+1)(C_2 + lceil \log_2 frceil)$$

For the other indexes, there are no t-locks and the number of leaf pages L' is simply:

$$L' = \lceil \frac{4 \cdot N + S \cdot N}{B} rceil$$

The cost of processing the index $cost_2'$ is then computed with this value of L'. Consequently, the total cost per operation, TOTAL in models 1 and 2 is given by:

$$\text{TOTAL} = P \cdot cost_1 + P \frac{F-1}{F} \left(cost_2 + (F_I - 1) \cdot cost_2'\right)$$

In models 3 and 4 there will be t-locks in all indexes and no distinction need be made for attribute z. Consequently, the total cost is:

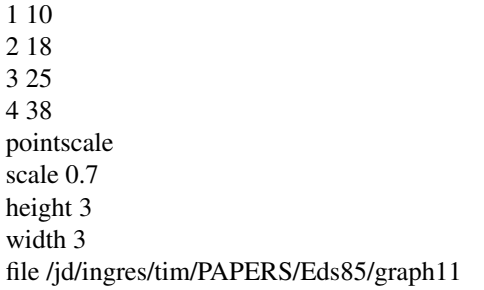
$$\text{TOTAL} = P \cdot cost_1 + P \cdot F_I \frac{F-1}{F} \cdot cost_2$$

5. PERFORMANCE RESULTS

In order to compare the six implementations we set the following parameters to constants as indicated:

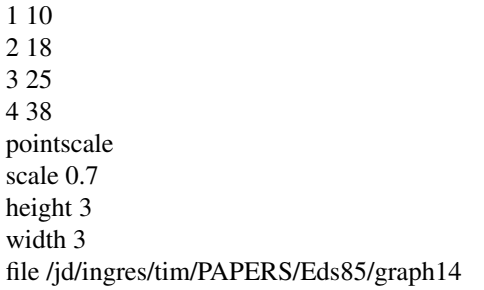
$C_1 = 10$
$C_2 = 30$
$B = 2000$
$W = 100$
$N = 1{,}000{,}000$
$F = 10$
$S = 10$
$F_I = 3$

One can interpret C_1 and C_2 as times in msec; the other parameters are typical of current applications. It can be noted that models 1 and 3 will yield the same expected total cost per basic operation for any particular setting of the model parameters. Hence, they differ only in what values of Q are intuitively reasonable. The same comment applies to models 2 and 4. For both pairs of models we varied the expected number of rules which cover a tuple, $t \cdot Q$ and the update probability, P. In all cases, predicate indexing performance is sensitive to the total number of rules, t, and we set that to 10000. Figure 1 plots expected cost per operation, TOTAL for models 1 and 3 for $t \cdot Q = 1$ as P is varied from 0 to 1 (EPI, EBL and EBBL stand for early predicate indexing, early basic locking and early boolean basic locking respectively. The others are similarly noted).



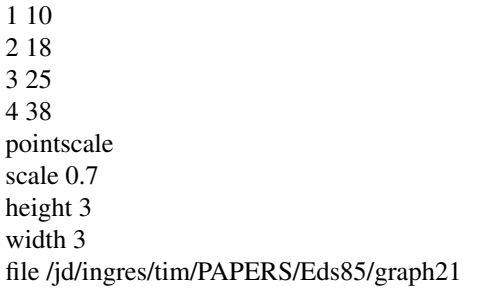
Models 1 and 3 : Costs vs Update Probability
Figure 1

For models 2 and 4 we set $t \cdot Q$ to be 15 to simulate each tuple being covered by 15 rules and performed the same analysis as above. The results appear in Figure 2.

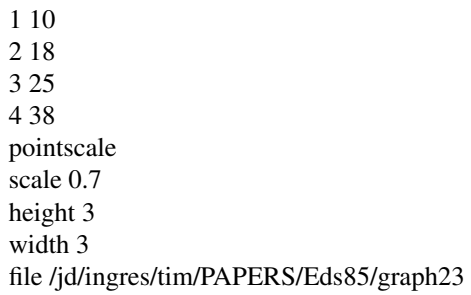


Models 2 and 4 : Costs vs Update Probability
Figure 2

<<<starting here -- will need some work>>>>> Notice that BL is the dominant alternative in both situations except when the update probability is low. Also, PI is never preferred for these parameters settings. On the other hand, when higher values of $t \cdot Q$ are utilized, then PI is always better than BL. Lastly, when the probability of update is low, there are situations where reduced marking is an attractive alternative. Figures 3 and 4 present such a case in which P is fixed at 0.15 and the number of rules which cover a tuple, $t \cdot Q$ is varied from 1 to 40.



Models 1 and 2 : Costs vs Expected Number of Predicates
Covering a Single Tuple (P = 0.15)
Figure 3



Models 3 and 4 : Costs vs Expected Number of Predicates
Covering a Single Tuple (P = 0.15)
Figure 4

Notice that reduced marking is the dominant strategy over the parameter space explored.

In general, predicate indexing offers superior performance when a tuple is covered by a large number of rules and when the probability of update is high. Basic locking must access and check a predicate for each t-lock in a tuple. This requires one random disk access per t-lock, whereas the qualifying predicates are clustered in the predicate index. This absence of clustering dooms basic locking to poor performance when the number of t locks becomes sufficiently large. On the other hand, the reduced marking system incurs substantial overhead when tuples are updated, dooming it to fail in update intensive environments.

On the other hand, basic locking excels when the expected number of rules which cover a tuple is low and reduced marking is the clear choice when the probability of update is low. Figure 5 gives the areas in the $(t \cdot Q, P)$ space where each of the implementations seems to work best.

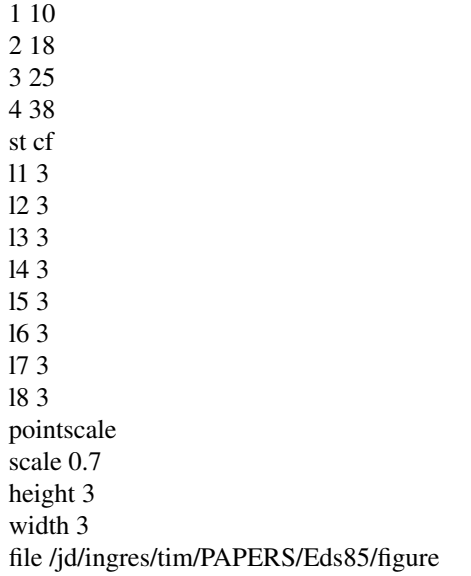


Figure 5

6. OTHER APPROACHES

It is clear from the preceding section that the best choice varies with the expected environment. Hence, it is feasible (although not very attractive) to implement more than one of the options and then choose an implementation based on the expected number of rules which cover a given tuple and the update probability.

Moreover, there are considerations concerning the generality of predicates required. If one needs predicates which include joins, then the predicate indexing schemes do not work, and a locking scheme must be employed. One possible composite scheme which overcomes this deficiency is now presented. A multi-relation predicate can be decomposed into a query plan which includes selections, projections, and a specific join algorithm run on pairs of relations (e.g. [WONG76, SELI79]). One could insert any selection or projection subquery in the plan into the appropriate predicate index. Then one could physically mark any pairs of tuples that satisfied the join clauses. It is possible that such a scheme may be advantageous. Other schemes to extend predicate indexing to join predicates are a subject for future research.

Also, if one restricts the predicates to model 1, then the predicate indexing scheme degenerates to a conventional B-tree which can be integrated with the secondary indexing mechanism. Hence, the implementation difficulty is eased in this special case.

Another improvement to basic locking and boolean basic locking is to organize the collection of rules so that the rules are clustered in a way similar to the predicate index described above. This will improve the performance of basic locking in environments with many rules covering the average tuple because it will cluster rules which must be checked together onto a smaller number of disk pages.

In addition, basic locking can be improved if additional complexity is tolerable. We will briefly describe another more sophisticated marking scheme which we term "mark storage" that may offer superior performance. The primary motivation behind the mark storage algorithm is to avoid searching every index of R every time a tuple is updated (as must be done in the reduced record marking algorithm). In a conventional database system without rule processing, only indexes used in the query plan must be read, and only those on updated attributes must be modified. The mark storage algorithm requires no extra I/O.

Consider the following general form of a predicate:

P: p_1 **and** $\cdots$ **and** p_k

Here, p_j is a restriction term such as "R.A = 5" or "50≤R.A≤100", and $1 \le k \le F$. The attributes for which a predicate has a restriction term will be denoted by a_{i_j}, where $1 \le j \le k$, and i_1 through i_k are the indexes of the attributes with a restriction term. We say that a predicate *partially matches* a tuple if at least one predicate term p_j, $1 \le j \le k$, matches an attribute a_{i_j} of the tuple. For example, consider the following relation and predicate:

R (A, B, C) -- indexes on A and B, no index on C

contents:
t: <A=5, B=10, C=14>

D_1: A=4 **and** C=15
D_2: B=10 **and** C=15
D_3: A=5 **and** B=10

Given this relation and set of predicates, we would say that D_2 partially matches t on attribute B, D_3 partially matches t on both A and B, and D_1 has no partial matches for t. In general, a predicate P *matches* a tuple if and only if the tuple partially matches P on all k terms of P. Attributes for which P does not have a term need not be considered. Thus, D_3 matches t, and D_1 and D_2 do not. Furthermore, if a predicate does not partially match a tuple on all terms, then it definitely does not match the tuple.

Given this background, the mark storage algorithm can be described. First, the algorithm requires every predicate to have at least one term on an indexed attribute. As in the reduced record marking scheme, each rule sets locks in all indexes for which it has a restriction term. Besides just locking the indexes, a list of rule id's is stored on every indexed attribute of every tuple in the database. A rule-id is stored on a tuple attribute if and only if that rule partially matches the tuple on that attribute **and** the attribute has an index. Associated with each rule-id stored with an attribute is information giving the positions of the attributes for which the rule predicate has a restriction term. This information can be stored with each rule-id as a bit string $b_1 \cdots b_F$ where b_i is 1 if the rule has a restriction term on attribute i, and 0 otherwise. In the example

database t will have the following lists of marks and associated bit strings on its attributes:

t: < A=5;[<D_3, 110>], B=10;[<D_2, 011>,<D_3,110>], C=15 >

Notice that attribute C of t has no list of rule-ids or bit strings since R.C has no index.

When a tuple is inserted into a relation, the lists of markings for the tuple attributes must be determined. This is easily accomplished since all indexes must be consulted, and the marks can be determined at low cost when the indexes are searched. Furthermore, if the tuple is modified with a **replace** command, the list of marks on an attribute changes only if that attribute was modified. This means that only indexes corresponding to updated attributes need be inspected and appropriate markings recomputed. This will result in a smaller update cost than the reduced marking algorithm.

The set S of predicates that *might* match a tuple, t are determined by finding the set of all <rule-id,bit-string> pairs on the tuple such that for every attribute of R which has an index, and for which the bit-string element b_i is a 1, an identical <rule-id,bit-string> pair occurs on the mark list for attribute i. For example, examining t as shown above, it is not possible to rule out a match on t for D_3 since b_1 and b_2 are 1 and $< D_3, 110 >$ is on the list for A and B. However it is possible to rule out a match on D_2 since $b_i = 0$ for D_2, but $< D_2, 011 >$ is not on the mark list for attribute A. The collection of predicates which cannot be ruled out must be checked just as in the basic locking algorithm.

This scheme may offer good performance because it avoids unnecessary I/O to indexes compared to the reduced marking algorithm and reduces the number of predicates that must be accessed and checked compared to the basic locking scheme. An analysis of the cost of this scheme is a subject of future research.

7. CONCLUSIONS

We have presented alternate implementations for rule indexing in a data base system. Our first proposal resembles predicate locking and uses a variation of R-trees to index the rule set. The other two approaches resemble versions of physical locking.

Our performance analysis results show that it is not possible to choose one implementation to support efficiently any rule based environment. Physical marking seems the most promising because of its ease of implementation, performance in simple environments, and extensibility to join predicates. We have also proposed extensions to predicate indexing and reduced marking that attempt to overcome their disadvantages. Analysis of these schemes and investigation of other extensions are a topic of future research.

8. REFERENCES

[ESWA75] Eswaran, K., *"A General Purpose Trigger Subsystem and Its Inclusion in a Relational Data Base System"*, IBM Research, Report No RJ 1833, San Jose, CA, July 1976.

[ESWA76] Eswaran, K.P., J.N. Gray, R.A. Lorie, I.L. Traiger, *"The Notions of Consistency and Predicate Locks in a Database System"*, Communications of the ACM, (**19**), 11, 1976.

[GRAY78] Gray, J.N., *"Notes on Data Base Operating Systems"*, IBM Research, Report No RJ 2254, San Jose, CA, August 1978.

[GUTT84a] Guttman, A., *"R-Trees: A Dynamic Index Structure for Spatial Searching"*, Proceedings of the 1984 ACM-SIGMOD International Conference on Management of Data, Boston, MA, June 1984.

[GUTT84b] Guttman, A., *"New Features for Relational Database Systems to Support CAD Applications"*, PhD Thesis, University of California, Berkeley, June 1984.

[IOAN84] Ioannidis, Y., et al., *"Enhancing INGRES with Deductive Power"*, Position Paper, Proceedings of the 1st International Workshop on Expert Data Base Systems, Kiawah Isl., SC, October 1984.

[ROBI81] Robinson, J. T., *"The K-D-B tree: A Search Structure for Large Multidimensional Dynamic Indexes"*, Proceedings of the 1981 ACM-SIGMOD International Conference on Management of Data, April 1981.

[SELI79] Selinger, P. et. al., *" Access Path Selection in a Relational Database Management System"*, Proceedings of the 1979 ACM-SIGMOD International Conference on Management of Data, Boston, MA, June 1979.

[SELL86] Sellis, T. et. al., "An Analysis of R^+-Trees," (in preparation).

[STON75] Stonebraker, M.,*"Implementation of Integrity Constraints and Views by Query Modification"*, Proceedings of the 1975 ACM-SIGMOD International Conference on Management of Data, San Jose, CA, June 1975.

[STON84] Stonebraker, M. et. al., *"QUEL as a Data Type"*, Proceedings of the 1984 ACM-SIGMOD International Conference on Management of Data, Boston, MA, June 1984.

[STON85a] Stonebraker, M., *"Triggers and Inference in Data Base Systems"*, Proceedings of the Islamorada Expert Data Base Conference, February 1985 (to appear in Springer-Verlag book).

[STON85b] Stonebraker, M. et. al., *"Extending a Data Base System with Procedures"*, UC Berkeley, Memo No. UCB/ERL/M85/59, Berkeley, July 1985.

[ULLM85] Ullman, J., *"Implementation of Logical Query Languages for Data Bases"*, Proceedings of the 1985 ACM-SIGMOD International Conference on Management of Data, Austin, TX, May 1985.

[WONG76] Wong, E., Youssefi K., *"Decomposition: A Strategy for Query Processing"*, ACM Transactions on Database Systems, (**1**) 3, September 1976.

[YAO78] Yao, A. C., *"On Random 2-3 Trees"*, Acta Informatica, (**9**), 2, 1978.